Invasive Plant Detection With

Convolutional Neural Networks

Aditya Gupta

Adlai E. Stevenson High School | Grade 10

Computer Science | Design

Table of Contents

Acknowledgements

I would like to thank my family, especially my older brother, for their support in pursuing this research, and for their suggestions. I would also like to thank my sponsor, Dr. Brontman for dedicating her time to helping me implement this design project and for her feedback, which has greatly helped me improve this project.

Problem

The rapid globalization over the last century has improved connectedness and supplied several benefits to humans, but also introduced an inundation of complex problems into today's world. Most noticeably, the environment has been burdened tremendously by economic development, and the rise of invasive species has contributed to the damage sustained by the natural world. Invasive species are aggressive, nonindigenous organisms that throw an ecosystem out of balance and harm surrounding property, habitats, and native fauna and flora. They are now the leading cause of biodiversity loss, have threatened food security, and are a hindrance to economic growth; the Department of Environmental Services (n.d.) estimates invasive species cost the United States alone more than $120 billion every year. Invasive plants are a particular problem as monocultures can also lead to massive wildfires and the lack of public knowledge often leads to humans helping cultivate and spread invasive plants.

Currently, one of the largest obstructions to dealing with invasive plants is the limited amount of accurate tracking and reporting of the current locations and predicted spread of invasive plants. By using convolution neural networks (CNNs)—a deep learning technique to classify images—the goal of this design project is to develop an accurate algorithm that identifies if a given image of a plant contains an invasive or native plant and which species, which can then be applied to a mobile application or satellite imaging system to predict accurate trends of plants, and thus prevent the spread of invasive plants. The rise in precision and accuracy of convolutional neural networks over recent years makes it an ideal option for tracking invasive plants.

Design Criteria

Several design criteria for the convolutional neural network are applicable, listed below:

- The model should accurately determine the subclass (invasive, native) of plant in the image with an accuracy of at least 90%. The standard for the base accuracy of a prototype binary classification convolutional neural network is typically around 85–95%.
- The top-1 testing accuracy for the exact species of the plant should be greater than 80%. Top-K accuracy means that the correct classification of an image is in the K top predictions the model makes. For top-1 accuracy, that means the actual species of plant is the first prediction the model makes. Historically, the goal top-1 testing accuracy for a small scale convolutional neural network is typically around 70–85%. (Nielsen, 2015).
- After training, the model should require minimal computation processing and identify an image within a less than a second (on a modern device) and should be practical to use.
- The model should support a variety of images and conditions, including lighting, angle, saturation, and resolution. Non-professional images should still be able to generate good results from the model.
- The model should have minimal overfitting and apply to a broad range of real-world and testing data.

Review of Literature

The last few decades have seen a monumental amount of progress on the frontier of data science and artificial intelligence (AI). Computational advancements in processing power and data availability have allowed AI-based applications to skyrocket in efficacy and performance, making it vital to many services today. The broad applicability of AI to a variety of sectors such as medical imaging and public safety have empowered today's society and helped resolve many issues related to the man-made world. (Statistical Analysis System, n.d.). Now, researchers are applying these sciences to the natural world, hoping to gain new understandings and help undo the damage caused by humans. Connecting artificial programs to the natural world presents a set of challenging and unique problems that not only require a thorough understanding of related topics, but also significant amounts of iteration and optimization to design a practical solution. (Jones & Easterday, 2022).

Machine Learning and Deep Learning

Machine learning (ML) is a subset of artificial intelligence that uses data analysis to automate construction of an analytical model. A machine learning model learns from examples to find patterns and trends in the data it is presented and make good decisions with minimal human intervention. Often a machine learning model is asked to predict a value or classify something based on a set of inputs, and the goal is to output a suitable result given this set of inputs. (Statistical Analysis System, n.d.).

To do this, the model goes through a training phase where it is taught the relationship between inputs and outputs through many examples of what outputs would be accepted for a set of inputs. Using algorithms, the model develops an "understanding" of how the inputs affect the output. Each datapoint a machine learning model receives during training adds to its experience and can then be used to base future predictions on; this is known as supervised learning, which is the most common branch of ML. (IBM Cloud Education, 2020a).

In addition, there are several other branches of machine learning including unsupervised learning, semi-supervised learning, and reinforcement learning. The methods above are all widely used for different problems, but each require manual feature extraction or specialized processing done by the developer; for each data point, a specialist must manually annotate pull information out of the source data before it can be given to the model to analyze. This process is intensive and laborious, and often dependent on the ability of humans to do so. (Statistical Analysis System, n.d.). One area, known as deep learning (DL), focuses on removing the manual extraction step and allows the model to learn directly from raw data. The model must automatically learn to extract features from the data rather than having a human do it. This allows the model to learn “end-to-end” and perform tasks automatically. (MathWorks, n.d.).

Unlike traditional machine learning methods, deep learning scales well with data; traditional models often reach a limit in accuracy after a certain amount of training, but deep learning algorithms usually take longer to plateau. Deep learning often exceeds human capabilities and produces accurate and consistent results. (MathWorks, n.d.). Deep learning typically requires more computing power and energy than traditional machine learning models, which have been bottlenecks in the past. However, advances in the modern-day gaming industry have demanded high-end graphical-processing-units (GPUs) that have supplied these computational tools. Today, deep learning is the heart behind applications such as fraud detection, language translation, and even art generation. (Hardesty, 2017).

Neural Networks

Deep learning is primarily implemented with neural networks (NNs), an approach to artificial learning inspired by the human brain. A neural network is composed of layers, each of which contains nodes or neurons. These neurons are connected to nodes in the layer beneath and above them; using information from the previous layer, they output a value based on their activation function, which is then passed to the next layer to be used as information to repeat the process. (Hardesty, 2017). After many layers are chained together, the neural network reaches a conclusion or final output. When data is passed through the neural network like this, it is known as forward propagation. (IBM Cloud Education, 2020a).

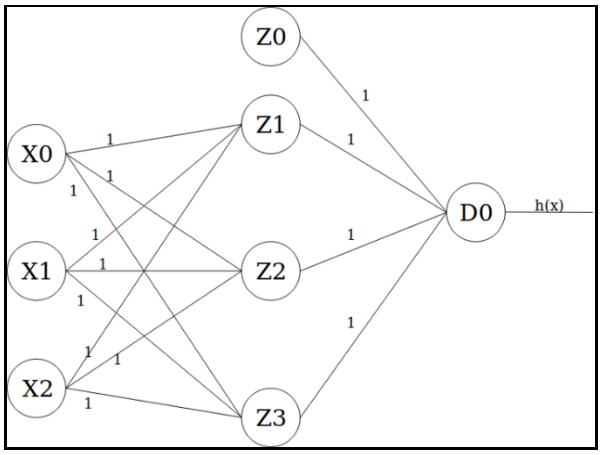


Figure 1. Simplified neural network (Al-Masri, 2019).

The first layer (X), or input layer, is where data is received, and the final layer (D), or output layer, is where the classification or prediction is delivered; these are the visible layers of the model. All layers in between are known as hidden layers. (Reyes, 2022). Traditionally, neural networks have only had 2–3 hidden layers, but modern hardware has enabled them to have up to 150. In fact, that is where the term “deep” in deep learning originated—the depth of neural networks in their number of hidden layers. (MathWorks, n.d.).

The perceptron was the earliest type of artificial neuron invented. A perceptron takes in inputs from earlier neurons and assigns a weight (w) to each input. Then, it multiplies the value of each input with the input's corresponding weight and sums the values together. If the sum exceeds a threshold value, the perceptron outputs 1; otherwise, the perceptron outputs 0. The threshold is typically referred to as the bias (b), where b-threshold (Nielsen, 2015).

$$\text{output } = \begin{cases} 0;\ \Sigma_j w_j x_j + b \leq 0 \\ 1; \Sigma_j w_j x_j + b > 0 \end{cases}$$

Figure 2. Perceptron activation function (Nielsen, 2015).

This function is known as the perceptron's activation function. In a sense, weights can be thought of as the importance of information from that neuron; higher significance means a higher weight. Similarly, bias can be thought of as how easy it is to satisfy the condition to output 1. Perceptrons can also be used to implement common bitwise operators like AND, OR, and NAND; together, these operators are universal for computation, which means any computational problem can theoretically be solved by a neural network. (Nielsen, 2015).

The neural network must also automatically determine the weights and biases—known as the parameters—to use for optimal accuracy. Ideally, a change in parameters would result in a relative change to the output. This would allow an algorithm to predict how changes to the parameters would affect the output and determine how to tune the parameters for the network so that the output is accurate or "learn" how to respond. (Nielsen, 2015).

Unfortunately, perceptrons do not behave in a predictable way; a minor change in the weights could cause the output of a perceptron to completely flip, resulting in uncontrollable changes to the output that make it difficult to tune the parameters. Consequently, modern neural networks don't use the perceptron's activation function but often use the sigmoid function, which outputs a decimal value between 0 and 1 while still using the same system of weights and biases. (Nielsen, 2015).

$$\frac{1}{1 + \exp\left(-\Sigma_j w_j x_j - b\right)}$$

Figure 3. Sigmoid neuron activation function (Nielsen, 2015).

The sigmoid function may seem quite different from the perceptron's function mathematically, but sigmoid is essentially a "smoothed-out" version of the perceptron's activation function (which is just a step function). As sigmoid is continuous, minor changes in the weights affect the output in a proportional way. This means that there is predictability in the neural network, and thus the neural network can be fine-tuned by an algorithm. (Nielsen, 2015).

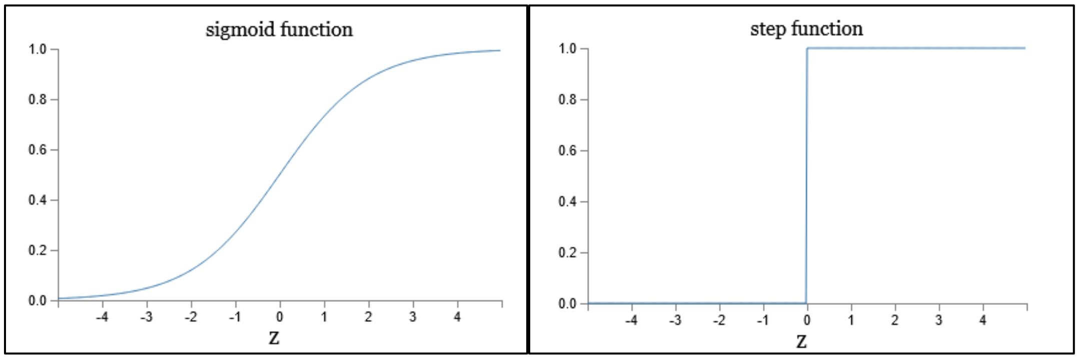


Figure 4. Sigmoid and step function graphs (Nielson, 2015).

Several functions work simultaneously to optimize the parameters of a neural network. Firstly, the neural network must know how inaccurate its predictions are to improve. After each epoch, or iteration of the training data set, a cost function calculates the total loss of the model, or the model's error rate. Accuracy is avoided because accuracy ignores the confidence of the model. (Al-Masri, 2019). Loss functions vary on the activation function, penalization desired, and output of the model. The goal of a neural network is to minimize the value of the cost function, somewhat like finding a line of best fit in a scatterplot. (IBM Cloud Education, 2020b).

Skewed inputs cause skewed outputs, so the loss of one layer affects the next. Thus, the loss of an output is the aggregated loss of all prior neurons. To reduce the loss, the model adjusts the parameters of neurons responsible for the most loss and penalizes their importance (weight). To do that, the network calculates the total loss at each neuron from the output to the input layer; losses are calculated in this order because calculations for a node can be reused to calculate loss for earlier nodes. This is known as backpropagation (Al-Masri, 2019).

The optimization function determines how parameters should be changed to improve performance of the network. Most commonly, this function is gradient descent, a tool of multivariable calculus to minimize the cost function. Gradient descent uses the derivative of the activation function and a learning rate to change parameters. A learning rate is the step taken to reach the point of convergence, or the point when loss is minimized. Typically, the learning rate is a small value (0.1); higher learning rates can reach the point of convergence in less epochs, but also risk overshooting the minimum loss and behave less consistently than small learning rates. Once the cost function reaches near zero, the neural network has essentially reached the point of convergence and stopped learning (IBM Cloud Education, 2020b).

Training the model excessively results in its overfitting to the training data. The network starts to memorize training data after many epochs; the loss may continue to decrease, but the model becomes useless when applied to anything besides training data. To prevent overfitting, a validation data set is fed to the model after each epoch. If the neural network's loss for the validation set is less than previous epochs, the model continues training; otherwise, the model stops learning. The model cannot be overfit to validation data because the parameters are not adjusted on validation data, so the model has no "memory" of the data. (Koehrsen, 2018).

The success of a model is dependent on its activation function; a poor activation function, such as the perceptron, makes it hard for gradient descent to tune the parameters of a neural network. Similarly, a linear function cannot be used as an activation function due to a constant slope (and a useless derivative). Although sigmoid allows for backpropagation to adjust the parameters with moderate accuracy, it still has some drawbacks (Baheti, 2022).

Sigmoid is not zero-centered, resulting in all the weights having the same sign when tuned by gradient descent. For certain situations, this makes training inefficient. However, the larger issue is that sigmoid has horizontal asymptotes at 0 and 1, which results in the derivative of the function becoming inconsequential at large absolute values. Consequently, a neuron can have significant changes in its weights while still having its output barely changed, known as the vanishing gradient problem. Several other activation functions are often used in favor of sigmoid due to this, such as the Tanh and ReLU (Rectified Linear Unit) functions (Baheti, 2022).

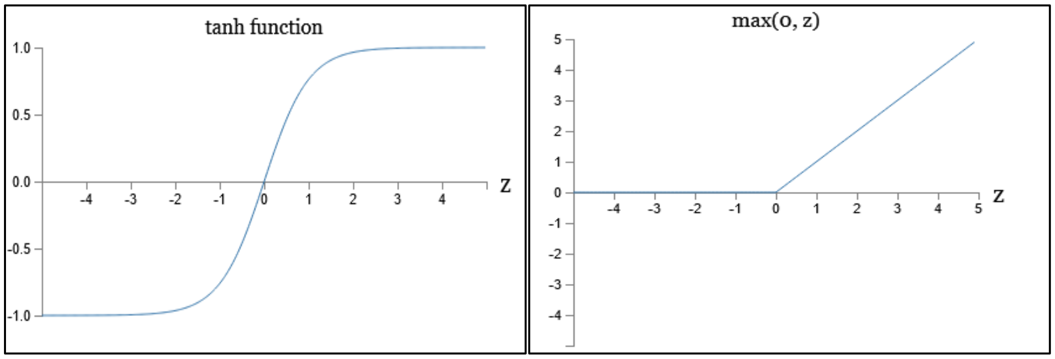


Figure 5. Tanh and ReLU function graphs (Nielsen, 2015).

Although Tanh is zero-centered, it still suffers from the vanishing gradient problem. ReLU is often preferred as it alleviates this problem, but it is not zero-centered. Many variants such as the ELU, “Leaky” ReLU, and Google’s swish function are also used. Ultimately, every activation function is situational and has benefits and drawbacks. (Baheti, 2022).

Convolutional Neural Networks

Convolutional neural networks (CNNs) are a special type of neural network that are used for classifying images and helping machines see the world as humans do, which is known as the field of computer vision. Computers interpret images with pixels, where each pixel has a red, blue, and green (RGB) color channel with a value representing the amount of that color. To interpret images, CNNs convert images to three-dimensional tensors, or arrays, where two dimensions indicate the position of a pixel, and the third holds RGB values. (Mandal, 2021).

Images are converted to a tensor during the preprocessing stage, when data is rescaled to a fixed resolution, cleaned, and data augmentation is applied. Data augmentation is the process of artificially increasing the size of a dataset by randomly rotating, horizontally flipping, or cropping images. The idea is that the position and orientation of an object in an image does not change the object itself. Each epoch, the transformations applied are slightly different to the image, so the model is not fed the exact same image in each epoch. Through these small variations, the hope is that the model learns the true attributes of the object and is not biased to a position or orientation for it. (Koehrsen, 2018).

Images hold significant amounts of data; for example, a 360 x 360-pixel image would have 129,600 pixels, and each pixel has three RGB values, resulting in 388,800 input values. With a traditional neural network, the following layer would require 388,800 weights per neuron, which requires absurd computing power, enables overfitting, and is not scalable. (Gavrilova, 2021).

CNNs use a special mathematical operation known as convolution, which essentially merges two datasets together to compress them while still retaining as much information as possible. CNNs use filters and slide them over the input data, multiplying the values of each pixel by the weight in that position of the filter, and then sums the values of each pixel in the filter together. Each filter applied is a single convolutional layer, and multiple convolutional layers apply different filters and produce different feature maps. A feature map is essentially a compressed tensor that highlights the important characteristics of the image. (Gavrilova, 2021).

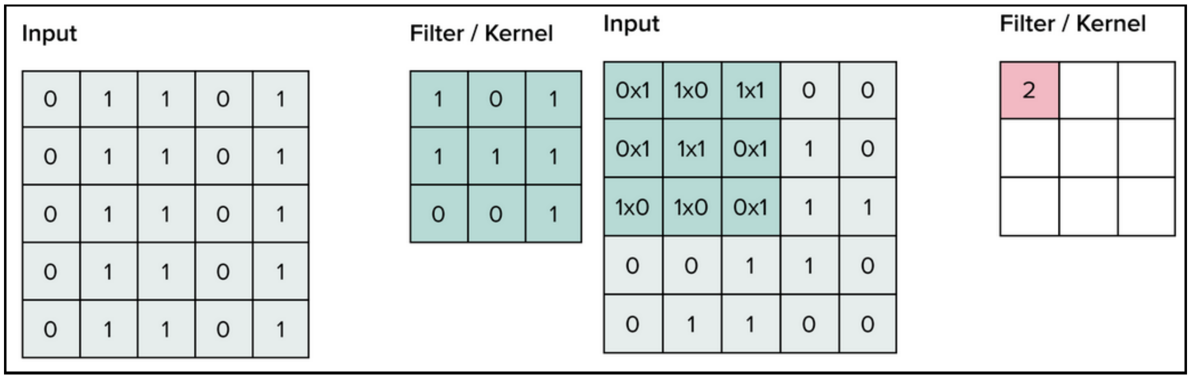


Figure 6. Convolution operation (Gavrilova, 2021).

As seen in figure 6, the filter shrinks the size of the input when sliding over it. To prevent this, "padding" or fake pixels are often added to the edge to keep the original image intact. The filter does not need to slide over the image one pixel at a time. Instead, it can skip pixels by a certain stride, so the filter does not overlap. This compresses the size of the image with marginal costs in accuracy, although a stride of one is most common. CNNs also share weights between neurons, so each neuron can be assigned fewer weights. (Cornelisse, 2018).

Once feature maps are produced by several convoluted layers, they get combined in the pooling layers of the network. The pooling layers compress feature maps into a summarized output; the idea is to take the average or maximum value of an N x N region in the image, effectively compressing the size of the feature map. This reduces training time and removes noise, since tiny variances in the image which may affect feature maps get removed when pooling the information, improving stability of the CNN. (Gavrilova, 2021).

After pooling the information, the output is flattened to one-dimension and fed to the fully connected layers, which operate just like a traditional neural network. They make sense of the extracted features and classify images. The result is the CNN's calculated probability that an image belongs to each of the classes. Accuracy for CNNs is measured in top-k accuracy, where the accuracy measures the occurrences for which the correct class was in the top k most likely classes the CNN predicted. Generally, benchmarks for CNN prototypes are a top-1 accuracy of greater than 85% and a top-5 accuracy greater than 95%. (Cornelisse, 2018).

The process of developing a convolutional neural network from scratch takes copious amounts of computing power and has high hardware requirements. Even with industry leading technology and hardware, these models can take weeks to train. (Reyes, 2022). As a result, training neural networks from scratch is quite rare, and other methods, such as transfer learning, are more favorable. The idea behind transfer learning is to take a pre-trained model trained on unrelated data and specialize it for a specific application. (MathWorks, n.d.).

The goal is to "transfer" general skills from the old model and add new skills to optimize and fine-tune the model to one task. For example, this can mean keeping the convoluted layers that extract universal features from images, but adding new skills specific to the task, and altering the fully connected layers later in the model to classify images more precisely for a subset of categories. By leveraging the general feature extraction process of a pre-existing model, the new one does not need to be trained as extensively on common tasks. (Koehrsen, 2018). Transfer learning has been widely successful. CNNs traditionally need a lot of data and training, and transfer learning makes them more accessible, leading to more widespread feature learning, high accuracy, and computational efficiency of CNNs. (Gavrilova, 2021).

The Environment and Invasive Plants

In many ways, artificial intelligence is an asset to environmental preservation and restoration. The usage of computer vision to enable self-driving cars can significantly cut down on carbon emissions by up to 50% by optimizing travel routes and cutting down total distance traveled, which can help reduce climate change. Using AI to optimize farming and agriculture can yield much larger harvests and help strengthen food security. Predicting weather patterns and natural disasters such as forest fires can save human lives and prevent ecological destruction. However, AI also has drawbacks; training a professional-grade neural network can release up to 250,000 pounds of carbon dioxide into the air. (Jones & Easterday, 2022).

Considerable damage to the environment is also caused by invasive species. Invasive species are overly aggressive non-indigenous organisms that swarm and harm a region they were introduced to; U.S. agricultural crops are not native to the area but are not invasive since they do not harm surrounding property, ecosystems, or native fauna. Often, invasive species reproduce rapidly with few predators. (National Geographic, n.d.).

Invasive species vary based on the region, but in regions where they are considered invasive, they are parasitic to the environment. Often, they can hunt down endangered prey or outcompete native organisms for food, such as silver carp in the Missouri River. Invasive species are the leading cause for biodiversity loss such as when brown tree snakes caused the extinction of nine out of 11 native bird species on the island of Guam. In the Chesapeake Bay and Gulf Coast of the United States, large rodents known as nutria have grazed much of the native wetland grass vital for shelter, food, and preventing soil erosion. This has destroyed the natural habitat, causing millions of dollars of damage. (National Geographic, n.d.).

Invasive plants are often especially problematic. For instance, water hyacinth is an invasive plant that has damaged ecosystems across the globe—including the entire southern United States—by crowding out most other plant life, causing biodiversity loss. It has even damaged the economy, growing so dense that ports had to be shut down in Uganda because boats couldn't get through. By blocking sunlight from reaching underwater, plants and algae died out causing a decline in the fishing industry (National Geographic, n.d.).

Since plants form the basis for an ecosystem, when non-native plants dominate a region, it helps even more invasive species to take hold. Even worse, many flammable invasive species such as cheatgrass and salt cedar supply fuel for wildfires to rampage on in less fire-prone ecosystems. These wildfires kill off more native plants and trees, allowing invasive plants to spread further, which creates a feedback loop. Invasive plants also generally anchor less soil than native plants, circulating less water through soil which leaves soil depleted of nutrients and causes desertification. Weeds can threaten food security and cause agricultural losses, contributing to global hunger. (Department of Environmental Services, n.d.).

More than 6,500 nonindigenous species currently inhabit the United States, although approximately 1,000 species are considered invasive and cause significant harm to the environment. (DeMarco, 2015). The United States sustains an estimated $120 billion in economic damage annually because of these invasive species, and more than 270 are invasive plants. Efforts to control invasive species have been mostly unsuccessful. Using chemicals or biological agents often harm native species. Introducing new organisms to remove an invasive species often backfires, causing a new invasive specimen. Methods that have worked have been expensive and often taken years. (Department of Environmental Services, n.d.).

The issue mostly originates from difficulty in predicting how invasive species will spread. For example, zebra mussels were unknowingly attached to the hulls of cargo ships; they now clog pipes in the United States. Furthermore, the average person doesn't know which species are harmful, often contributing to the spread of invasive species and particularly invasive plants, since humans often cultivate nonindigenous plants by accident; water hyacinth is grown for its colorful flowers. In fact, cultivation by humans is one of the chief reasons invasive plants have been so successful (DeMarco, 2015).

Poor reporting of invasive plants leads to slow responses, which are a major hinderance to dealing with invasive plants effectively. For example, at Lake Minnetonka and Houghton Lake in Minnesota and Michigan, respectively, Eurasian watermilfoil (EWM) has overtaken the lakes and turned them into seed banks, due to the slow response to contain it. Now, the lakes are closed off to the public most of the year, with harsh chemicals and reapers cutting the invasive plants back to keep a small part of the lakes open in the summer. In contrast, Lake Leelanau responded quite quickly, and has been able to contain the spread of EWM. However, it still faces risk of closing and has already spent more than $6 million to contain EWM. (Tyra, 2022).

Use of deep learning can help limit the spread of invasive species, although it has not been largely explored. Champer et al. (2021) explored how to use supervised machine learning to model the genes of invasive rodents and help control their population, but the larger context of AI has been focused on simply identifying plants from images, or identifying plant disease, such as the study conducted by Borhani et al. (2022). Invasive plants are a significant problem due to difficult-to-predict spread patterns, and deep learning is a powerful tool to discover patterns. Utilizing this novel tool to control invasive plant spread will be the emphasis in this project.

Materials

- 64-bit computer with a CUDA-capable GPU running Windows, macOS, or Linux.
- Internet access
- Anaconda 3 Distribution with PyTorch
- JetBrains DataSpell IDE
- iNaturalist Dataset (Retrieved 1/8/23)

Design Plan

1. Download and configure Anaconda 3 Distribution.
2. Using conda installer, install PyTorch. If needed, install Pillow.
3. Download and configure DataSpell.

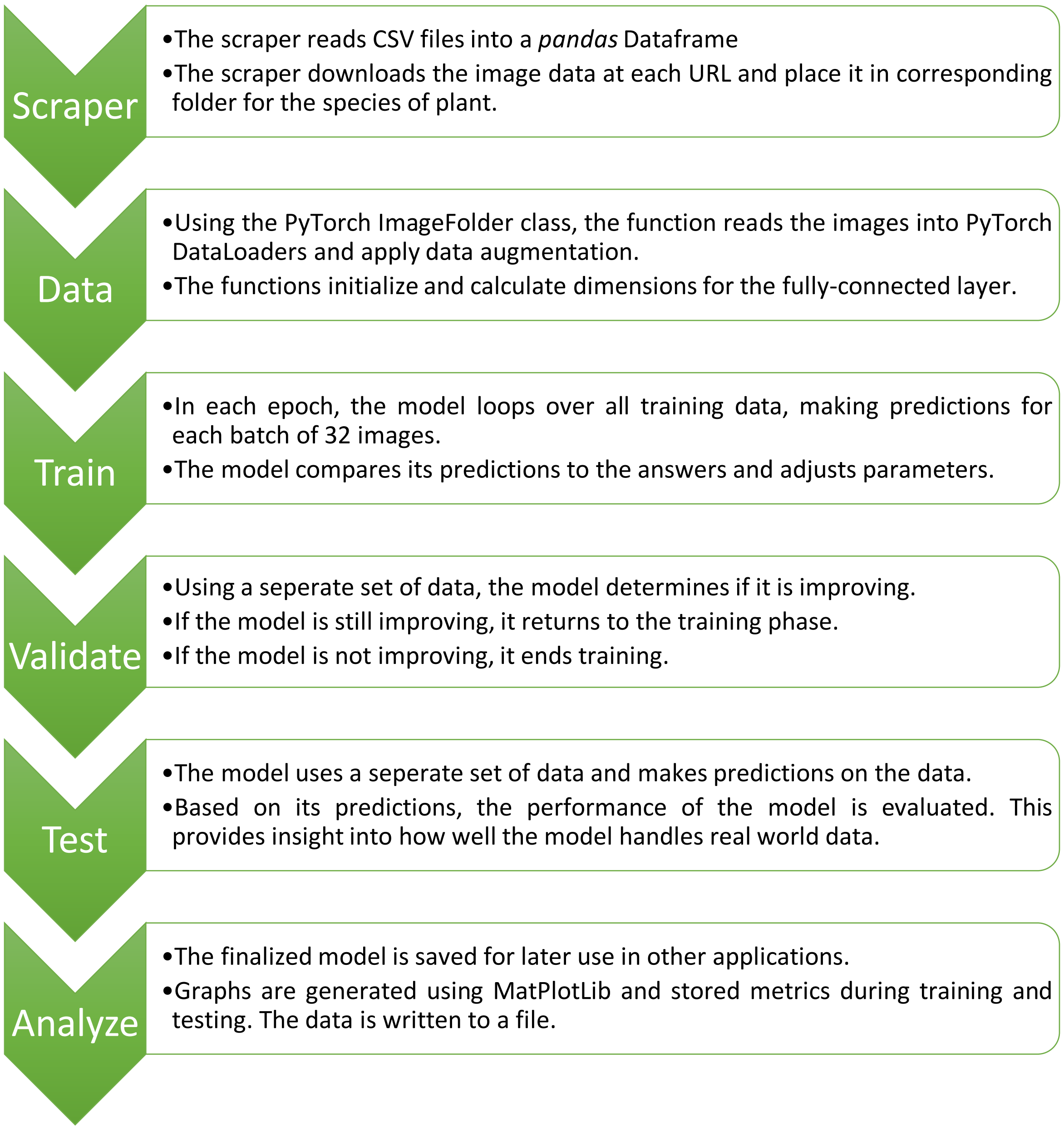
Configuring the Data

4. Develop a custom export configuration for each species using iNaturalist's export tool.
5. Queue the export for each plant based on each configuration (United States, Plant, Research Grade, Verified). An iNaturalist account is required.
6. Once available, download the CSV data file for each plant. This file has image URL data.
7. Use the *pandas* library (installed with Anaconda) to inspect and clean the CSV data.
8. Use inbuilt Python libraries and *pandas* to develop a scraper that automatically downloads and resizes images to a fixed resolution and organizes them in line with the PyTorch ImageFolder requirements. Images are hosted by iNaturalist at the URLs exported in step 6.

Developing the Source Code

9. Extend the neural network module of PyTorch by creating a child class to implement the convolutional neural network classifier.
10. Write the initialization function for the class, which implements each layer of the model, the optimizer and loss function, and maintains performance statistics. The design should be modular and extendable or inherited with transfer learning applied.
11. Using PyTorch Dataset and Dataloader, import the image data and convert it to PyTorch tensors. Split the data into train, test, and validation splits with a 70:15:15 ratio using a seeded generator for reproducibility.
12. Using PyTorch transforms, implement data augmentation.
13. Write the training function for the CNN class. It should have a training and validation phase, calculate accuracy at each batch, backpropagate, and early-stop when necessary.
14. Write the testing function for the CNN class. Use PyTorch's evaluation mode to prevent unexpected behavior and data leakage with batch normalization statistics.
15. Write code to generate and save graphs of the data training, validation, and testing data, as well as write that data to text files.
16. Create an instance of the CNN class and specify parameters.
17. Use PyTorch to train and evaluate the designed model. The designed model should automatically calculate the accuracy and loss. Export results as needed.
18. Analyze the results to improve the model and tune hyperparameters.
19. Repeat steps 17 to 18 until the design criteria is met and improvement is negligible.
20. Collect additional metrics for the finalized model.

Below, a diagram illustrates a rough layout of the program and model and how each part of the program and model works. The full code can be seen in *Appendix A: Full Code* or at https://github.com/aaggupta07/invasive-plants-cnn.


Scraper
•The scraper reads CSV files into a *pandas* Dataframe
•The scraper downloads the image data at each URL and place it in corresponding folder for the species of plant.
Data
•Using the PyTorch ImageFolder class, the function reads the images into PyTorch DataLoaders and apply data augmentation.
•The functions initialize and calculate dimensions for the fully-connected layer.
Train
•In each epoch, the model loops over all training data, making predictions for each batch of 32 images.
•The model compares its predictions to the answers and adjusts parameters.
Validate
•Using a seperate set of data, the model determines if it is improving.
•If the model is still improving, it returns to the training phase.
•If the model is not improving, it ends training.
Test
•The model uses a seperate set of data and makes predictions on the data.
•Based on its predictions, the performance of the model is evaluated. This provides insight into how well the model handles real world data.
Analyze
•The finalized model is saved for later use in other applications.
•Graphs are generated using MatPlotLib and stored metrics during training and testing. The data is written to a file.

Results and Discussion

The model was designed and tested over several versions, each of which had the same input data: 69,850 images of plants, classified over 23 main plant species. The generator splitting data into training, testing, and validation sets was seeded to increase reproducibility.

The first version had 6 convolutional layers, each with batch normalization and ReLU activations, as well as a linear fully connected layer. Images were rescaled to a size of 64x64 pixel resolution, and the model had a constant learning rate of 0.001 and batch size of 32. It was trained with a drop-off delta of less than 2% improvement in the training loss over the last 3 epochs of training. Due to drop-off, training ended after the 10 epochs. Results are below.

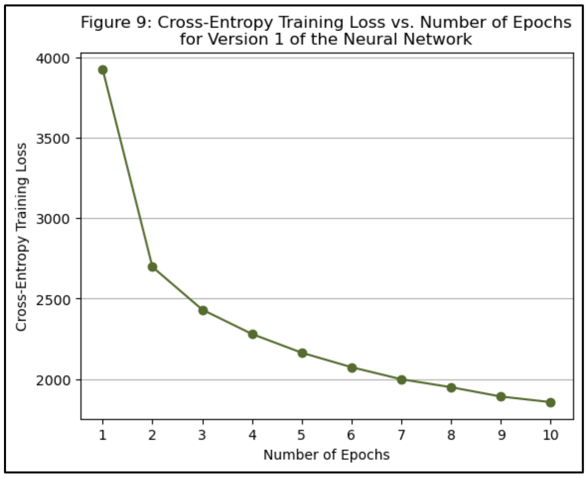
Figure 7: Cross-Entropy Loss After Each Epoch for Version 1 of the Neural Network

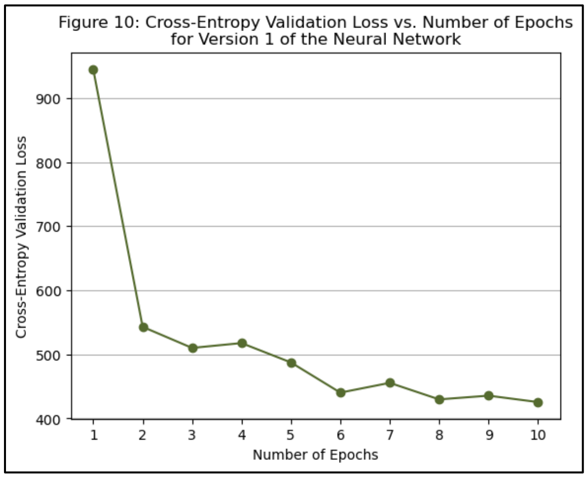
Cross-Entropy Loss	Phase	
Epoch	**Training**	**Validation**
1	3924.2022	944.8808
2	2697.0819	543.0168
3	2431.9490	510.1149
4	2280.4494	517.6754
5	2164.1564	487.5107
6	2074.3520	440.3706
7	1999.6317	455.6339
8	1950.0473	429.8155
9	1892.3094	435.6476
10	1857.4659	425.6118
Test		434.7267
Average	2327.1645	519.0278
Best	1857.4659	425.6118

Figure 8: Accuracies After Each Epoch for Version 1 of the Neural Network

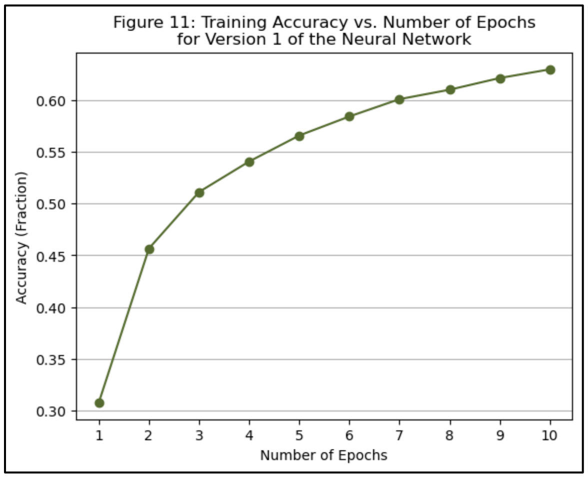
Accuracy (%)	Phase	
Epoch	**Training**	**Validation**
1	30.8085	29.6494
2	45.6569	49.3807
3	51.1057	51.9436
4	54.0447	52.3152
5	56.5504	55.4021
6	58.3858	58.8319
7	60.0556	58.5271
8	60.9651	60.1848
9	62.0892	58.9367
10	62.9251	61.3377
Test		60.0419
Average	54.2587	54.2319
Best	62.9251	61.3377

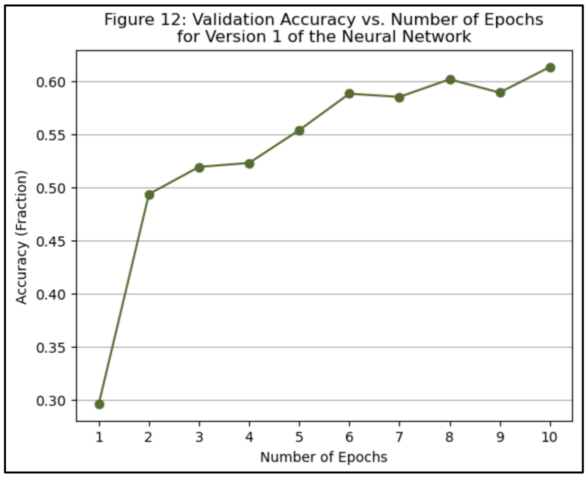
Validation loss is smaller than training loss, as only 15% of the data is used for validation and 70% for training, so the aggregate validation loss is smaller. Comparing the two, one expects about 4.67 times as much loss in training compared to validation, which is well represented.





Both the training and validation losses showed declines to less than half the initial loss over the training period, although a significant portion of improvement was just between the first and second epochs. Training loss (Figures 7 & 9) continued to decrease throughout the 10 epochs, but validation loss (Figures 7 & 10) failed to improve significantly beyond 6 epochs, resulting in early stopping. This also suggests that the model was overfitting beyond 6 epochs. Overfitting occurs when the model starts to memorize the answers to training data, and as a result, becomes a lot worse when applied to unseen (test) data. The validation dataset is used to prevent this from happening. More graphs are below.





The prediction that the model began to overfit is further reinforced by Figures 11 and 12; training accuracy continued to increase by about 1% per epoch after epoch 6 to about 62.93% (Figure 8), but the testing accuracy was just 60.04% (Figure 8), consistent with the best validation accuracy which plateaued after epoch 6 (Figure 12), achieving a best of 61.34% (Figure 8). Overall, the model performed poorly. However, the version 1 model performed much better than the baseline (control) for accuracy, which was 4.35%—the accuracy that a model randomly choosing a class achieves. Note that Figures 11 and 12 closely resemble mirror images of Figures 9 and 10, which is expected, as a lower loss generally results in a higher accuracy.

The second version of the model featured several improvements, including the addition of 3 additional convolutional layers to the model and enhancements to the parameters for drop-off to reduce the likelihood of overfitting. The largest change was scaling input images to a resolution of 244x244 pixels, rather than 64x64 pixels, which enables the model to distinguish between distinct species better. An example comparing the different resolutions is shown below.

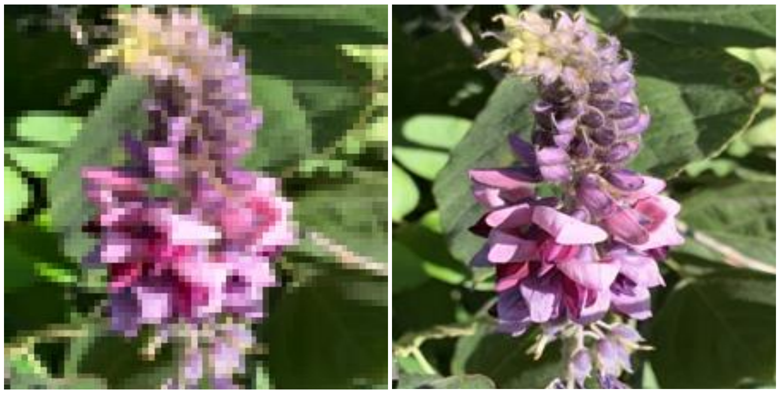


Figure 13. A kudzu plant in 64x64 pixel resolution (left) compared to a 244x244 pixel resolution (right).

Each adjustment was individually assessed and found to improve performance, and thus implemented in the model. Results for the second version of the model are shown below.

Figure 14: Cross-Entropy Loss After Each Epoch for Version 2 of the Neural Network

Cross-Entropy Loss	Phase	
Epoch	**Training**	**Validation**
1	2723.5195	555.5240
2	2114.6796	424.4887
3	1875.1780	382.7998
4	1721.1572	354.3189
5	1580.3352	340.0819
6	1465.3683	441.2573
7	1369.9641	299.7209
8	1288.5578	307.7074
9	1221.0141	281.2049
10	1158.2099	308.2335
Test		286.5857
Average	1651.7984	369.5337
Best	1158.2099	281.2049

Figure 15: Accuracies After Each Epoch for Version 2 of the Neural Network

Accuracy (%)	Phase	
Epoch	**Training**	**Validation**
1	44.9886	46.8750
2	57.0900	60.3944
3	62.2977	64.6818
4	65.4533	66.6254
5	68.2268	68.2355
6	70.3973	62.9478
7	72.3962	71.8464
8	73.6695	71.3129
9	75.2677	73.9901
10	76.3796	71.0461
Test		73.1993
Average	66.6167	65.7955
Best	76.3796	73.9901

Figure 16: Cross-Entropy Training Loss vs. Number of Epochs for Version 2 of the Neural Network

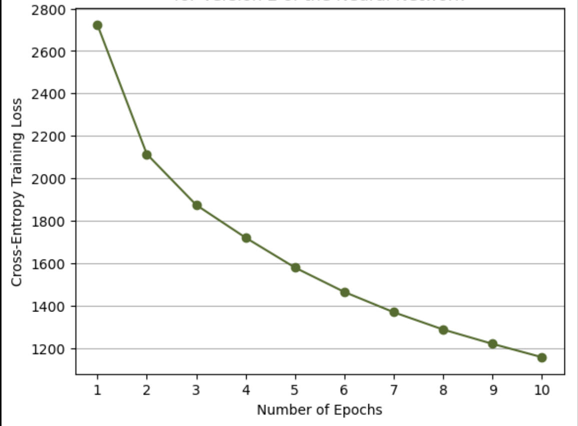


Figure 17: Cross-Entropy Validation Loss vs. Number of Epochs for Version 2 of the Neural Network

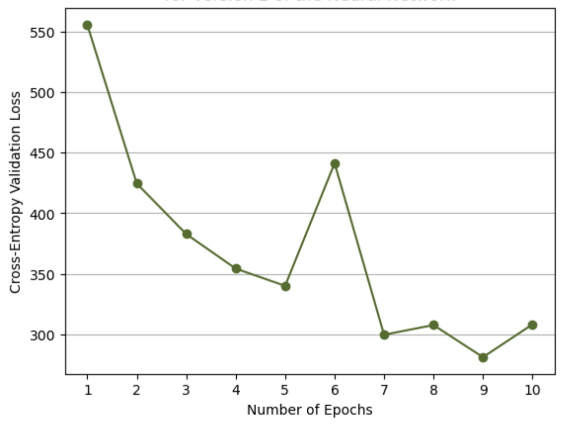


Figure 18: Training Accuracy vs. Number of Epochs for Version 2 of the Neural Network

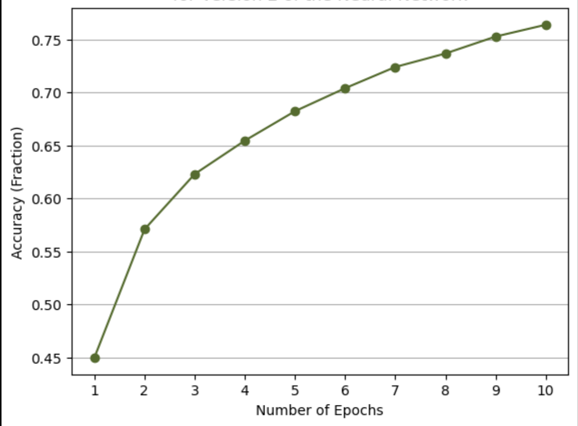
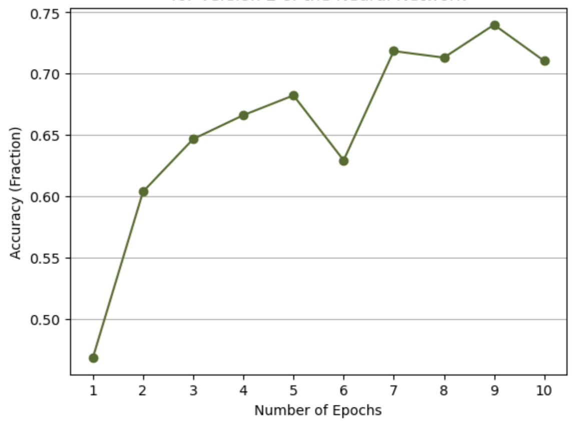


Figure 19: Validation Accuracy vs. Number of Epochs for Version 2 of the Neural Network



The model performed better, with a lower training and validation loss in each epoch compared to version 1 (Figures 14, 16–17). Note that this model also had early stopping invoked after 10 epochs to prevent overfitting. Epoch 6 was likely an outlier during validation and was not concerning due to the model following its trend immediately afterwards (Figures 17 & 19). Compared to the first version, final validation loss decreased by approximately 144.41 units, and validation accuracy improved from 61.33% to 73.99% (Figures 14–15). The final testing accuracy was 73.20%, consistent with the validation accuracy and an improvement over the first version.

Although the changes improved accuracy, they made the model extremely slow and computationally expensive to train; the model took almost 10 times longer to train, taking 46 minutes, 21 seconds to train on a modern GPU. Furthermore, it was still well below the desired accuracy. To alleviate this, the third version of the model used a transfer learning approach, using the ResNet-50 classifier as the base model, thus completely overhauling the model architecture. Initially, weights were only updated for the fully connected layer.

Figure 20: Cross-Entropy Loss After Each Epoch for Version 3 of the Neural Network

Cross-Entropy Loss	Phase	
Epoch	Training	Validation
1	858.5990	127.6261
2	539.2303	112.7065
3	468.5301	106.5356
4	433.9134	104.2562
5	408.6926	101.0891
6	389.9710	101.9395
7	376.9138	102.9549
Test		106.8518
Average	496.5500	108.1583
Best	376.9138	101.0891

Figure 21: Accuracies After Each Epoch for Version 3 of the Neural Network

Accuracy (%)	Phase	
Epoch	Training	Validation
1	69.5670	77.6105
2	79.6569	80.0019
3	81.9404	80.6498
4	83.1556	80.9261
5	83.7500	81.4405
6	84.4996	81.2691
7	84.7998	80.9165
Test		80.5926
Average	81.0528	80.4021
Best	84.7998	81.4405

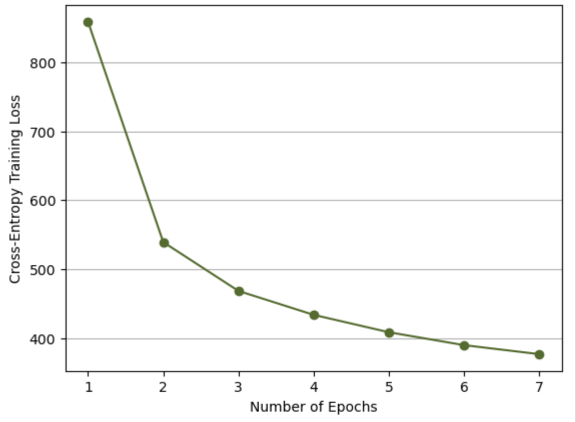


Figure 22: Cross-Entropy Training Loss vs. Number of Epochs for Version 3 of the Neural Network

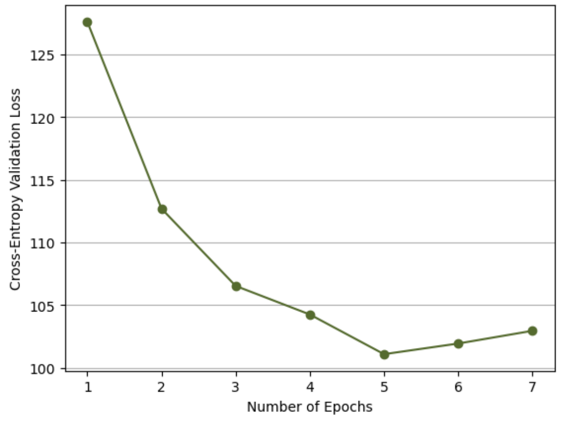


Figure 23: Cross-Entropy Validation Loss vs. Number of Epochs for Version 3 of the Neural Network

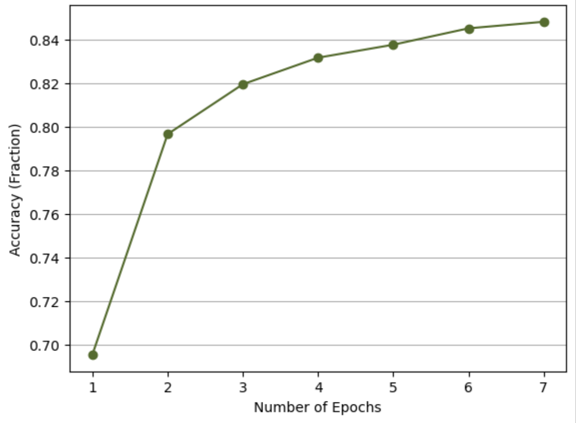


Figure 24: Training Accuracy vs. Number of Epochs for Version 3 of the Neural Network

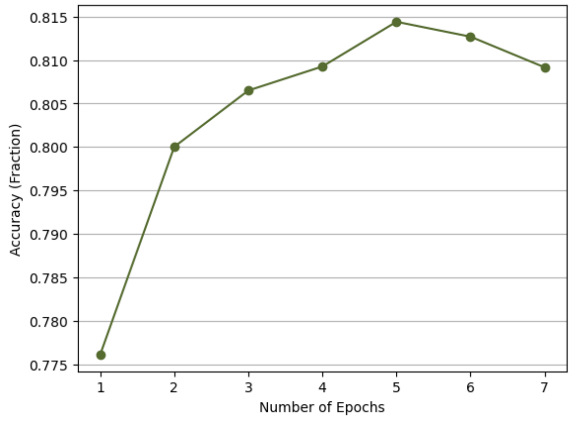


Figure 25: Validation Accuracy vs. Number of Epochs for Version 3 of the Neural Network

Overall, this version marked a considerable improvement to the earlier versions, with the training loss decreasing to 376.91 units (Figure 20), a considerable improvement over the previous 1158.21 units (Figure 14). Furthermore, validation loss decreased from 281.20 units to 101.09 units (Figures 14 & 20). Testing accuracy improved from 73.20% to 80.06%, again consistent with the best validation accuracy of 81.44%. The model also learned faster, early stopping after seven epochs, which improved its efficiency over earlier models. The changes not only improved accuracy but sped up the model training time to approximately 20 minutes.

This met the top-1 accuracy in the design criteria, but the model could still be improved. For the fourth version, all layers were trained, not just the fully connected layer. In addition, various hyperparameters were tuned through various trials—such as the learning rate and batch size—regularization was added, and optimizations were made to increase efficiency.

Figure 26: Cross-Entropy Loss After Each Epoch for Version 4 of the Neural Network

Cross-Entropy Loss	Phase	
Epoch	**Training**	**Validation**
1	1120.5395	149.1066
2	649.9461	162.9531
3	513.6902	121.1393
4	432.6292	128.5977
5	364.5975	121.4354
Test		128.0314
Average	616.2805	136.6464
Best	364.5975	121.1393

Figure 27: Accuracies After Each Epoch for Version 4 of the Neural Network

Accuracy (%)	Phase	
Epoch	**Training**	**Validation**
1	77.9819	85.8899
2	87.0136	84.7180
3	89.5704	88.7481
4	91.1053	88.6623
5	92.4031	89.2435
Test		88.1955
Average	87.6149	87.4524
Best	92.4031	89.2435

Figure 28: Cross-Entropy Training Loss vs. Number of Epochs for Version 4 of the Neural Network

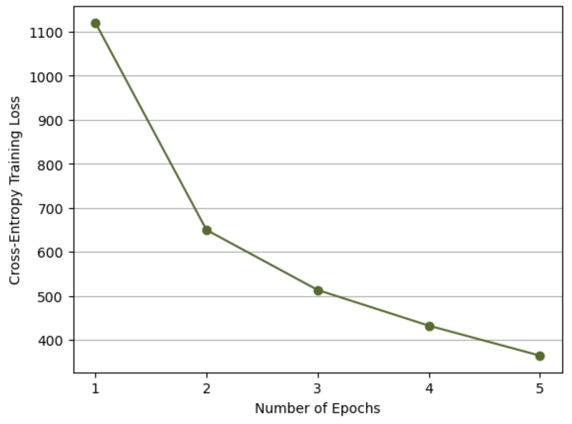


Figure 29: Cross-Entropy Validation Loss vs. Number of Epochs for Version 4 of the Neural Network

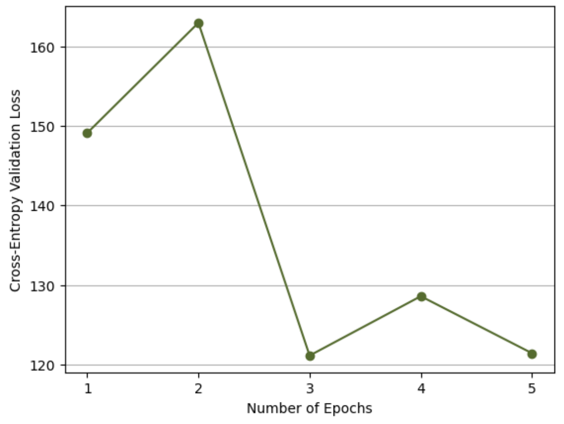


Figure 30: Training Accuracy vs. Number of Epochs for Version 4 of the Neural Network

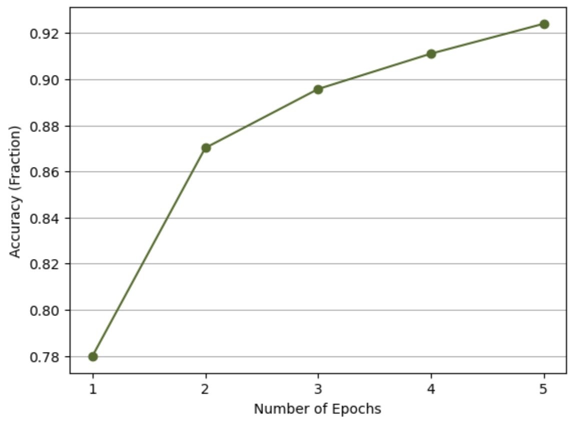
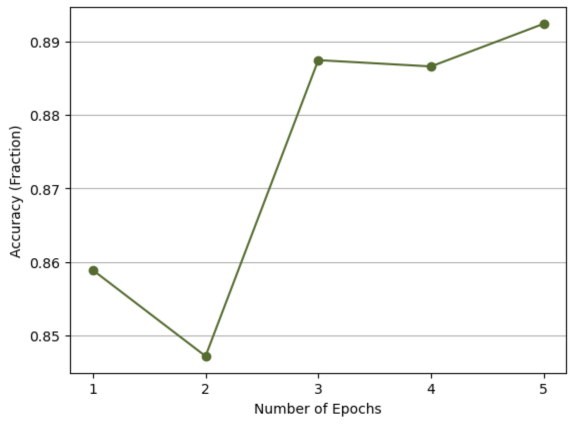


Figure 31: Validation Accuracy vs. Number of Epochs for Version 4 of the Neural Network



The graphs show improvements again from the previous version, with the model having maximally learned over five epochs. Furthermore, it displayed rather significant improvements in accuracy and loss over previous versions, earning an 85.89% validation accuracy after just one epoch, a higher validation accuracy than achieved during the best epoch for any of the previous versions. The model essentially learned very quickly over the initial epoch. During epoch 2, the model was likely traversing local maxima, resulting in a spike in validation loss. Note that the y-axis is also expanded on Figures 28–31, so fluctuations are smaller than they appear. Although it had a slightly higher validation loss than version 3, this version also had a significantly larger number of trainable parameters, so the loss per layer was significantly lower than version 3. This is reflected in the final validation and testing accuracies, which were 89.24% and 88.20%, respectively.

Additional metrics were collected for version 4, finding that it distinguished the subclass of plants as native or invasive correctly 94.74% (meeting the design criteria). In testing, the model classified 10,478 images in approximately 30 seconds, classifying about 349 images per second or an image every 2.86ms. Below, two more graphs summarize testing data. Refer to data tables above for exact testing result values if needed.

Figure 31: Testing Loss Across All Versions of the Neural Network

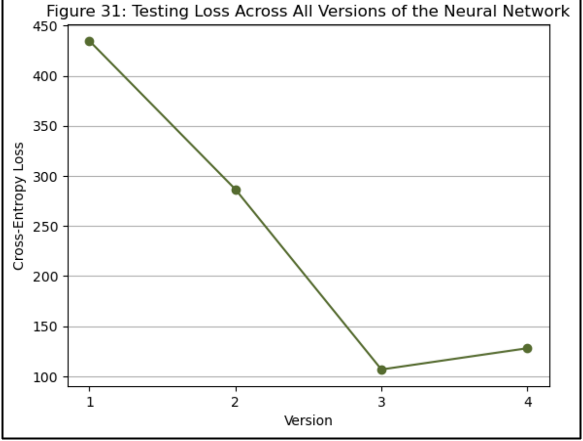
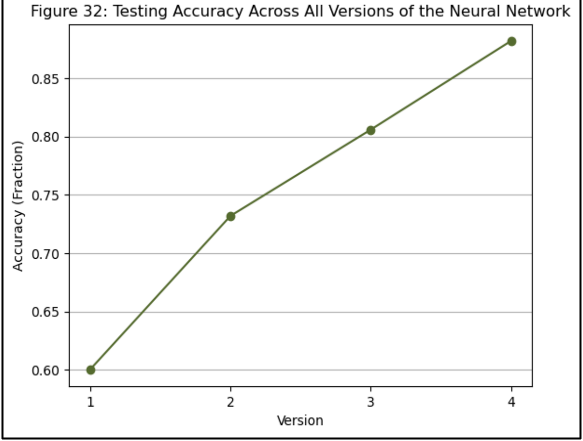


Figure 32: Testing Accuracy Across All Versions of the Neural Network



As mentioned, version 4 had significantly more trainable parameters accounting for the slight increase in testing and validation loss in Figure 32. Figure 33 shows a steady increase in accuracy over all four versions, verifying that the model did indeed improve over each version. Below, figure 34 summarizes validation and testing accuracies per epoch for each model version.

Figure 34: Validation Accuracies After Each Epoch for Each Version of the Neural Network

Accuracy (%)	**Version**			
Epoch	**1**	**2**	**3**	**4**
1	29.6494	46.8750	77.6105	85.8899
2	49.3807	60.3944	80.0019	84.7180
3	51.9436	64.6818	80.6498	88.7481
4	52.3152	66.6254	80.9261	88.6623
5	55.4021	68.2355	81.4405	89.2435
6	58.8319	62.9478	81.2691	
7	58.5271	71.8464	80.9165	
8	60.1848	71.3129		
9	58.9367	73.9901		
10	61.3377	71.0461		
Test	60.0419	73.1993	80.5926	88.1955
Average	53.6509	65.7955	80.4021	87.4524
Best	61.3377	73.9901	81.4405	89.2435

Figure 34 shows the percent validation accuracies of each version of the model over each epoch, average and best validation accuracies, as well as the test accuracies, rounded to four decimal places. Because data splits were seeded, the results should be reproducible with small error margins; deviation from these results should be at most one percent. Note that although multiple changes were made in each version, each change was individually and jointly tested and found to improve overall model accuracy, but results were grouped together for simplicity.

Conclusion

This design project aimed to solve a widespread environmental conservation problem: the lack of accurate reporting and detection of invasive plants, which causes slow responses to combat those invasive plants. Now, more than ever, these invasive plants threaten to cause biodiversity loss, habitat destruction, food shortages, and large economic damages, among other issues, as noted by the Department of Environmental Services (n.d.). Controlling invasive species requires large-scale systems with consistent and accurate identification, requirements that convolutional neural networks excel at, which inspired the approach for this project.

The first design did not meet design criteria, as it achieved a testing accuracy of only 60.04%. This resulted in many changes such as scaling the input data to different dimensions, adopting transfer learning architecture, and adjusting hyperparameters which led to the final design meeting all design criteria initially set: The model classified plants down to the species with a top-1 testing accuracy of 88.20%, exceeding the criterion for 80%. The model correctly identified a plant as invasive or native 94.74% of the time, fulfilling the requirement for 90%; overfitting was minimized with a validation dataset. The model also classified images fast—it took roughly 2.86ms to classify an image. Although this timing is based on a batch of 10,478 images, the time should not be noticeably slower on individual images. The dataset used features images taken by ordinary people; images are then research certified through a collective verification system, so the model excels at working with the “average” image of a plant that is taken.

One source of experimental error comes from the data since anyone can take an image of a plant and upload it to the iNaturalist database. iNaturalist has verification tools so that once enough people identify a plant, it becomes certified as “Research Grade.” This reduces the amount of bad data, but it should be expected that a small amount of the data probably influenced the model in negative ways, which could occasionally lead to errors in the real world. The use of data augmentation and scaling images to a downsized resolution (224x224 pixels) helped minimize the consequences. Additionally, as with any neural network, some inherent randomness is present, based on values of initial weights and data splits. A seed was used to increase reproducibility, but minor fluctuation in results will still be present.

The most notable real-world application of this research is that it can be used for invasive plant tracking and detection, to enable faster response times and reduce the ecological, environmental, and economic damage invasive plants cause. However, it can also be used for public awareness, since it enables people to find out more information about the plants they are growing and make sure they aren't contributing to the damage invasive plants cause—especially because of how lightweight the model is—and further reduce environmental damage.

One limitation of the model is its behavior when plants of conflicted species are placed within the image. Generally, the model tends to pick the image which occupies more pixels and space in the image, rather than the plant in the foreground, which is typically more practical. Additionally, the model still does misclassify about 1 in 10 images and is not inclusive of many lesser-known plant species. To address this, improvements are still being made to the model.

Specifically, development on expanding the number of identifiable plant species currently is being worked on, as well as designing a system to assign plants a threat level based on their history in the region and their propagation methods. In addition, collaboration with other individuals and teams—as well as integrating the model with other professional systems in place by the Department of Agriculture and Department of the Interior—is being explored.

Reference List

Al-Masri, A. (2019, January 29). *How does back-propagation in artificial neural networks work?* Towards Data Science. Retrieved October 8, 2022, from https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7

Baheti, P. (2022, October 3). *Activation functions in neural networks [12 types & use cases]*. V7 Labs. Retrieved October 10, 2022, from https://www.v7labs.com/blog/neural-networks-activation-functions

Borhani, Y., Khoramdel, J., & Najafi, E. (2022). A deep learning based approach for automated plant disease classification using vision transformer. *Scientific Reports*, *12*(1), 11554. https://doi.org/10.1038/s41598-022-15163-0

Champer, S. E., Oakes, N., Sharma, R., García-Díaz, P., Champer, J., & Messer, P. W. (2021). *Modeling CRISPR gene drives for suppression of invasive rodents using a supervised machine learning framework*.

DeMarco, E. (2015, January 30). *Invasive plants taking over the U.S.* American Association for the Advancement of Science. Retrieved October 30, 2022, from https://www.science.org/content/article/invasive-plants-taking-over-us

Cornelisse, D. (2018, April 24). *An intuitive guide to convolutional neural networks*. freeCodeCamp. Retrieved October 18, 2022, from https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

Department of Environmental Services. (2016, February 17). *The problem with invasive plants*. Environmental Services. Retrieved October 30, 2022, from https://www.portlandoregon.gov/bes/article/330681

Gavrilova, Y. (2021, August 3). *Convolutional neural networks for beginners*. Serokell Software Development. Retrieved October 11, 2022, from https://serokell.io/blog/introduction-to-convolutional-neural-networks

Hardesty, L. (2017, April 14). *Explained: Neural networks*. MIT News | Massachusetts Institute of Technology. Retrieved October 8, 2022, from https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.

Hunter, J. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90–95.

IBM Cloud Education. (2020a, May 1). *What is deep learning?* IBM. Retrieved October 1, 2022, from https://www.ibm.com/cloud/learn/deep-learning

IBM Cloud Education. (2020b, October 27). *What is gradient descent?* IBM. Retrieved October 9, 2022, from https://www.ibm.com/cloud/learn/gradient-descent iNaturalist community. (2022). Observations of invasive and native plants in the United States, observed between 2015–2022. Retrieved January 6, 2023, from https://www.inaturalist.org.

Jones, E., & Easterday, B. (2022, June 28). *Artificial intelligence's environmental costs and promise*. Council on Foreign Relations. Retrieved September 27, 2022, from https://www.cfr.org/blog/artificial-intelligences-environmental-costs-and-promise

Koehrsen, W. (2018, November 26). *Transfer learning with convolutional neural networks in Pytorch*. Towards Data Science. Retrieved October 9, 2022, from https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

Mandal, M. (2021, July 23). *CNN for deep learning: Convolutional neural networks*. Analytics Vidhya. Retrieved October 12, 2022, from https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

MathWorks. (n.d.). *What is deep learning? | How it works, techniques & applications*. MathWorks. Retrieved September 30, 2022, from https://www.mathworks.com/discovery/deep-learning.html

National Geographic. (n.d.). *Invasive species*. National Geographic Society. Retrieved October 30, 2022, from https://education.nationalgeographic.org/resource/invasive-species

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32 (pp. 8024–8035). Curran Associates, Inc. Retrieved from http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Reyes, K. (2022, September 30). *What is deep learning and how does it work [explained]*. Simplilearn. Retrieved October 1, 2022, from https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-deep-learning

Statistical Analysis System. (n.d.). *Machine learning: What it is and why it matters*. SAS. Retrieved September 30, 2022, from https://www.sas.com/en_us/insights/analytics/machine-learning.html

Tyra, E. (2022, September 2). How it started...how it's going: An invasive Eurasian watermilfoil update from Lake Leelanau. The Leelanau Ticker. Retrieved November 24, 2022, from https://www.leelanauticker.com/news/how-it-startedhow-its-going-an-invasive-eurasian-watermilfoil-update-from-lake-leelanau/

Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference (pp. 56–61).

Appendix A: Full Code

This appendix features the code for the implementation, training, and testing of the final (version 4) model, and the code written to write data to files and generate graphs. The code makes extensive use of the PyTorch library alongside other libraries. Code for all versions of the model, as well as a pretrained model, is available at https://github.com/aaggupta07/invasive-plants-cnn. All libraries used were cited in the reference list.

```
# Imports
import copy

import numpy as np
import matplotlib.pyplot as plt

# Torch imports
import torch
import torch.nn as nn
import torch.nn.functional as f
import torch.optim as optim

from torch.utils.data import DataLoader
from torch.utils.data import random_split

from torchvision import datasets, transforms, models

# For saving images
import pandas as pd
import requests
import shutil
import os
from PIL import Image

# Metrics
from tqdm.auto import tqdm
import time


# Get all the images from their urls, resize them, and save them locally
folder = "all-plant-data"
new_folder = "plant-images-64"
temp = os.path.join(new_folder, "temp")
rescale = 64
start = False
files = os.listdir(folder)

for file in tqdm(files, desc='Plants Completed: '):
    df = pd.read_csv(os.path.join(folder, file))
    plant = str(file)[:-4]
    for idx, url in enumerate(tqdm(df['image_url'], desc=f'{plant}: ')):
        attempts = 0

        while attempts < 5:
            try:
                filepath = os.path.join(new_folder, plant, str(idx))

                r = requests.get(url, stream=True)
                r.raw_decode_content = True

                with open(temp, 'wb') as f_out:
                    shutil.copyfileobj(r.raw, f_out)

                image = Image.open(temp)
                image = image.resize((rescale, rescale))
                image.save(str(filepath) + ".png")

                if attempts > 0:
                    print(f"Succeeded after {attempts + 1} tries")
                break

            except requests.exceptions.ConnectionError:
                print(f"ConnectionError: Could not get image {idx} of file
{file}")
                if attempts < 4: print("Attempting to retry...")
                else: print("Failed 5 times, moving on to next image")
                attempts += 1


# Create a mapping of Torch ImageFolder IDs to Native/Invasive Classes
is_native = set()
for file in os.listdir('native-plant-data'):
    is_native.add(str(file)[:-4])

subset_native = set()

for idx, file in enumerate(os.listdir('plant-images')):
    if str(file) in is_native: subset_native.add(idx)

print(subset_native)


# Transfer Learning with ResNet50
class ResNetClassifier(nn.Module):
    def __init__(self, lr, tol, batch_size, epochs, num_classes):
        super(ResNetClassifier, self).__init__()

        # Initialization
        self.batch_size = batch_size
        self.epochs = epochs
        self.num_classes = num_classes
        self.lr = lr
        self.tol = tol
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

```
        # For plotting
        self.loss_history = []
        self.acc_history = []
        self.epoch_acc_history = []
        self.val_loss_history = []
        self.val_acc_history = []

        self.test_acc = -1
        self.test_loss = -1
        self.sub_acc = -1

        # Model
        self.model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT,
download=True).to(self.device)

        self.in_dims = self.model.fc.in_features
        self.model.fc = nn.Linear(in_features=self.in_dims,
out_features=self.num_classes)

        # Optimizer and loss function
        self.loss = nn.CrossEntropyLoss()
        self.optimizer = optim.AdamW(self.parameters(), lr = self.lr)

        # Move model to the GPU and get data
        self.to(self.device)

        self.train_data_loader = None
        self.val_data_loader = None
        self.test_data_loader = None
        self.get_data()


    # Forward pass for the model
    def forward(self, batch_data):
        batch_data = batch_data.clone().detach().to(self.device)
        out = self.model(batch_data)
        return out


    # Get data to train the model
    def get_data(self):
        # Data augmentation
        tf = transforms.Compose([transforms.RandomHorizontalFlip(),
transforms.ToTensor()])

        invasive_data = datasets.ImageFolder('plant-images', transform=tf)

        train_data, val_data, test_data = random_split(invasive_data, [0.7,
0.15, 0.15], generator=torch.Generator().manual_seed(61))

        self.train_data_loader = DataLoader(train_data,
batch_size=self.batch_size, shuffle=True, num_workers=4)
        self.val_data_loader = DataLoader(val_data,
batch_size=self.batch_size, shuffle=True, num_workers=4)
        self.test_data_loader = DataLoader(test_data,
batch_size=self.batch_size, shuffle=True, num_workers=4)
```

```
    # Train the model
    def train_(self):
        stop = False
        time_started = time.time()

        best_model = copy.deepcopy(model.state_dict())
        best_loss = 10000000.0
        best_acc = 0
        past = 0

        for i in range(self.epochs):
            header = f"Epoch {i+1}/{self.epochs}"
            print(header)
            print("-" * len(header))

            loader = self.train_data_loader
            for phase in ['train', 'validate']:
                if phase == 'train':
                    self.train()
                else:
                    self.eval()
                    loader = self.val_data_loader

                # For plotting and statistics
                epoch_loss = 0
                epoch_acc = []

                # Main training loop
                for in_data, label in loader:
                    self.optimizer.zero_grad()

                    # Get model prediction and determine how well it did
                    label = label.to(self.device)
                    prediction = self.forward(in_data)
                    classes = torch.argmax(prediction, dim=1)

                    wrong = torch.where(torch.tensor(classes !=
label).to(self.device),
                                        torch.tensor([1.]).to(self.device),
                                        torch.tensor([0.]).to(self.device))

                    # Calculate loss and accuracy
                    acc = 1 - torch.sum(wrong) / self.batch_size
                    batch_loss = self.loss(prediction, label)
                    epoch_loss += batch_loss.item()
                    epoch_acc.append(acc.item())

                    # Train through backpropagation
                    if phase == 'train':
                        batch_loss.backward()
                        self.optimizer.step()

                        # Statistics
                        self.acc_history.append(acc.item())
                elapsed = time.time() - time_started
                print(f"Finished {phase} with loss {epoch_loss} and accuracy 
of {np.mean(epoch_acc)} in time "
                      f"{int((elapsed // 60) // 60)}:{int(elapsed // 
60)}:{int(elapsed % 60)}")

                if phase == 'train':
                    self.loss_history.append(epoch_loss)
                    self.epoch_acc_history.append(np.mean(epoch_acc))
                else:
                    self.val_loss_history.append(epoch_loss)
                    self.val_acc_history.append(np.mean(epoch_acc))
                    best_acc = max(best_acc, np.mean(epoch_acc).item())

                    if (best_loss - epoch_loss > 0.03 * best_loss and 
best_loss - epoch_loss > 4) or (best_loss - epoch_loss >= 50):
                        past = 0
                        best_loss = epoch_loss
                        best_model = copy.deepcopy(model.state_dict())
                    else:
                        past += 1
                        print("No improvement over last epoch.")
                        if past >= self.tol:
                            print("No improvement in model loss. Early 
stopping...")
                            stop = True
                            break

            print()
            if stop: break

        print("Training complete...")
        print(f"Best validation loss: {best_loss}")
        print(f"Best validation accuracy: {best_acc}")

        model.load_state_dict(best_model)
        return model


    def test_(self):
        self.eval()

        time_started = time.time()
        total_loss = 0
        total_acc = []
        total_sub_acc = []

        for in_data, label in self.test_data_loader:

            # Compare correct answers to model
            label = label.to(self.device)
            prediction = self.forward(in_data)
            classes = torch.argmax(prediction, dim=1)
            wrong = torch.where(torch.tensor(classes !=
label).to(self.device),
                                torch.tensor([1.]).to(self.device),
                                torch.tensor([0.]).to(self.device))

            wrong_sub = 0
            for i in range(len(classes)):
                if (classes[i].item() in subset_native) != (label[i].item()
in subset_native): wrong_sub += 1


            acc = 1 - torch.sum(wrong) / self.batch_size
            sub_acc = 1 - wrong_sub / self.batch_size
            batch_loss = self.loss(prediction, label)

            total_acc.append(acc.item())
            total_sub_acc.append(sub_acc)
            total_loss += batch_loss.item()

        self.test_acc = np.mean(total_acc)
        self.test_loss = total_loss
        self.sub_acc = np.mean(total_sub_acc)
        elapsed = time.time() - time_started
        print(f"Finished with loss {total_loss} and accuracy of
{np.mean(total_acc)} in {int(elapsed // 60)}:{int(elapsed % 60)}")
        print(f"Testing Subset Accuracy (Invasive/Native): {model.sub_acc}")


# Train the model and save it
model = ResNetClassifier(lr=0.001, tol=2, batch_size=32, epochs=10,
num_classes=23)
model = model.train_()
model.test_()
torch.save(model.state_dict(), 'model-v4')


# Generate graphs from model-saved data
v = input("Enter version #: ")
fig = int(input("Enter figure #: "))
epoch_dims = np.arange(1, len(model.loss_history)+1, 1)
path = 'graphs-analysis/'

# Graph 1
plt.plot(model.loss_history, marker='o', color='darkolivegreen')
plt.title(f"Figure {fig}: Cross-Entropy Training Loss vs. Number of
Epochs\nfor Version {v} of the Neural Network")
plt.xlabel("Number of Epochs")
plt.ylabel("Cross-Entropy Training Loss")
plt.xticks(np.arange(len(model.loss_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-train-loss-graph.png", bbox_inches='tight')
plt.show()

# Graph 2
plt.plot(model.val_loss_history, marker='o', color='darkolivegreen')
plt.title(f"Figure {fig+1}: Cross-Entropy Validation Loss vs. Number of
Epochs\nfor Version {v} of the Neural Network")
plt.xlabel("Number of Epochs")
plt.ylabel("Cross-Entropy Validation Loss")
plt.xticks(np.arange(len(model.val_loss_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-val-loss-graph.png", bbox_inches='tight')
plt.show()

# Graph 3
plt.plot(model.epoch_acc_history, marker='o', color='darkolivegreen')
plt.title(f"Figure {fig+2}: Training Accuracy vs. Number of Epochs\nfor
Version {v} of the Neural Network")
plt.xlabel("Number of Epochs")
plt.ylabel("Accuracy (Fraction)")
plt.xticks(np.arange(len(model.epoch_acc_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-epoch-acc-graph.png", bbox_inches='tight')
plt.show()

# Graph 4
plt.plot(model.val_acc_history, marker='o', color='darkolivegreen')
plt.title(f"Figure {fig+3}: Validation Accuracy vs. Number of Epochs\nfor
Version {v} of the Neural Network")
plt.xlabel("Number of Epochs")
plt.ylabel("Accuracy (Fraction)")
plt.xticks(np.arange(len(model.val_acc_history)), epoch_dims)
plt.grid(axis='y')
plt.savefig(path + 'v' + v + "-val-acc-graph.png", bbox_inches='tight')
plt.show()


def write_to(fw, arr, header):
    fw.write(header + '\n[' + str(arr[0]))
    for num in range(1, len(arr)):
        fw.write(', ' + str(arr[num]))
    fw.write(']\n\n')


try: v
except NameError: v = input("Enter version: ")

with open(f'graphs-analysis/v{v}-data.txt', 'w') as f_out:
    write_to(f_out, model.loss_history, "Training Loss")
    write_to(f_out, model.val_loss_history, "Validation Loss")
    write_to(f_out, model.epoch_acc_history, "Training Accuracy (Per Epoch)")
    write_to(f_out, model.val_acc_history, "Validation Accuracy")

    f_out.write(f"Testing Loss, Accuracy\n{model.test_loss},
{model.test_acc}\n\n")

    write_to(f_out, model.acc_history, "Training Accuracy (Per Batch)")
```